LABORATOR 1

Server şi client pentru transfer de fişiere

1 File Transfer Protocol (FTP)

File Transfer Protocol (“protocol de transfer fişiere” - ro) este un protocol folosit pentru a conecta două calculatoare din Internet astfel încât utilizatorul de pe un computer să poată transfera fişiere de pe celălalt computer într-un mod facil, foloisnd comenzi simple.

Mai exact, FTP este un protocol foarte utilizat pentru transferul de fişiere într-o reţea ce este construită pe baza TCP/IP, cum ar fi de exemplu reţeaua Internet.

Într-o tranzacţie FTP sunt implicate două calculatoare: un server şi un client. Serverul este cel ce ***ascultă*** în permanenţă un port (în mod implicit acesta este portul 21 pentru FTP), aşteptând solicitări de conectare din partea clienţilor. Un client este cel ce doreşte să se conecteze pentru a avea acces la fişierele aflate pe aceeaşi staţie ca şi serverul.

Imediat după ce un client s-a conectat, acesta poate efectua mai multe operaţii asupra fişierelor aflate pe server. Exemple de operaţii ce pot fi disponibile unui client conectat:

- Copierea de fişiere de pe staţia server pe staţia client (“*download*”, din punctul de vedere al clientului).
- Copierea de fişiere de pe staţia client pe staţia server (“*upload*” din punctul de vedere al clientului).
- Redenumirea de fişiere aflate pe server.
- Ştergerea de fişiere de pe server.

De vreme ce protocolul FTP reprezintă un standard bine definit, gratuit şi deja publicat, oricine poate să îşi creeze propriile aplicaţii server sau client care să îl implementeze.

Mai mult, un aspect demn de remarcat este acela că, odată ce un client a iniţializat conectarea la serverul FTP, acesta poate iniţializa transferul de fişiere, chiar dacă pe cele două calculatoare rulează sisteme de operare diferite.

Pentru transferarea datelor printr-o reţea folosind protocolul FTP sunt folosite de regulă două formate:

- modul **ASCII**
- modul **Binar**

Când modul de transmitere este **ASCII**, literele individuale, cifrele şi chiar caracterele sunt trimise folosindu-se pentru aceasta codurile lor ASCII. Staţia client va salva datele într-un format text specific sistemului de operare.

Astfel, vor apare diferenţe între modul de reprezentare al caracterelor line-feed şi carriage-return : în Linux de ex, **\n** este suficient pentru a semnaliza trecerea pe rând nou pe când pentru editorul de text implicit din Windows (Notepad) este necesară o combinaţie **\r\n** pentru a genera o trecere la rând nou.

Dacă modul de transmisie folosit este cel binar, arunci se vor transmite fişierele bit cu bit, într-un flux continuu de biţi.

2 Un client FTP simplu

2.1.1 Clasa URL

Pentru a implementa un client FTP simplu se va folosi clasa **`java.net.URL`** din Java. O instanţă din această clasă reprezintă de fapt un un indicator către o resursă web. O resursă web poate fi un director aflat pe un server FTP, dar poate la fel de bine fi un şir de interogare complex (*query*) adresat unui fişier php.

Clasa URL conţine mai multe metode, însă pentru început vom folosi doar câteva din acestea.

2.1.2 Metoda openStream()

Această metodă întoarce fluxul de intrare din care va fi posibilă citirea de date. Acest flux îşi are celălalt capăt la server, mai precis tocmai la resursa solicitată iniţial (în momentul iniţierii obiectului URL).

Documentaţia oficiala JavaAPI pentru această metodă este :

```
public final InputStream openStream()
                                throws IOException
```

Opens a connection to this URL and returns an InputStream for reading from that connection. This method is a shorthand for:

```
openConnection().getInputStream()
```

Returns:
an input stream for reading from the URL connection.
Throws:
`IOException` - if an I/O exception occurs.

Odată ce fluxul de intrare a fost deschis prin apelul openStream(), acesta va putea fi folosit pentru a iniţializa obiecte care să îl folosească pentru a citi din acesta. De ex : poate fi instanţiat un obiect **`BufferedReader`** pentru a putea citi linii de text din flux.

2.1.3 Client FTP funcţionând în mod text

În cele ce urmează va fi prezentat un client FTP foarte simplu, pe care îl veţi extinde pentru a-l transforma într-o aplicaţie flexibilă.

Acesta iniţializează un flux de citire şi apoi îl foloseşte pentru a citi linii te text, cu ajutorul unui obiect BufferedReader.

Fişierul este cel de la punctul 2.2. anexa 1.

Semnificaţia şirului de conectare folosit la instanţierea obiectului URL : "**ftp://admin:1234@80.96.123.145/test.txt**" :

- **admin** : acesta este numele de utilizator cu care se realizează autentificarea (*logarea*).
- **1234** este parola pentru utilizatorul specificat anterior
- **80.96.123.147** este adresa staţiei pe care rulează serverul
- **/test.txt** este fişierul solicitat: fişierul test.txt din radacina serverului

Serverul FTP la care se realizează conectarea a fost instalat în scop didactic pe staţia având adresa **80.96.123.145** şi va fi disponibil pe parcursul lucrării de laborator.

Mersul Lucrarii

1. Studiaţi exemplul dat (**clientul FTP minimal**).
2. Compilaţi şi lansaţi în execuţie clientul FTP oferit ca exemplu.
   Vizualizaţi rezultatele afişate la consolă.
3. Modificaţi şirul de caractere folosit pentru conectare, introducând valoarea:
   "**ftp://admin:1234@80.96.123.145/** "
   Recompilaţi şi apoi rulaţi din nou.
   Ce observaţi după rulare ?
   Ce va fi afişat în consolă ?
4. Modificaţi aplicaţia prezentată pentru a putea specifica din linia de comandă numele fişierului ce va fi solicitat Folosiţi pentru aceasta clasa **Sir** pusă la dispoziţie în laboratoarele anterioare. Testaţi apoi aplicaţia pentru alte fişiere text aflate pe serverul FTP pus la dispoziţie.
5. Modificaţi mai departe aplicaţia pentru a salva pe discul local fişierul obţinut de pe server. Pentru aceasta veţi folosi un obiect BufferedWriter. Mai precis, se va crea un obiect BufferedWriter prin apelul:

```
BufferedWriter bw =
new BufferedWriter( new FileWriter( "CALEFIS" ) );
```

   unde CALEFIS este numele complet cu care veţi salva fişierul pe disc.
   CALEFIS poate fi de exemplu:

   String strCale = "D:\\" + numeF ;

   Iar numeF este numele citit de la consolă.
6. Modificaţi primul exemplu (**clientul FTP minimal**) astfel: înlocuiţi obiectul **BufferedReader** cu un obiect **DataInputStream**. Acesta permite citirea de octeţi dintr-un flux, nu doar de şiruri de caractere:

```
DataInputStream dis = new DataInputStream(  is  );

byte byBuff[] = new byte[1024];

int nCit = dis.read( byBuff, 0 , 1024 );
System.out.println( "Citit: " + nCit + " octeti" );
```

   Eliminaţi cu totul bucla while – acum citirea s-a realizat într-un singur pas.

Folosind un obiect **DataOutputStream**, salvaţi într-un fişier conţinutul din bufferul byBuff. Acel fişier va reprezenta o copie la nivel de octet pentru fişierul de pe server.
Pentru scrierea în fişier folosind obiectul DataOutputStream veţi folosi metoda write, definită în documentaţia Java API ca:

```
public void write(byte[] b,
                  int off,
                  int len)
           throws IOException
```

Writes `len` bytes from the specified byte array starting at offset `off` to the underlying output stream. If no exception is thrown, the counter `written` is incremented by `len`.

Specified by:

`write` in interface `DataOutput`

Overrides:

`write` in class `FilterOutputStream`

Parameters:

`b` - the data.

`off` - the start offset in the data.

`len` - the number of bytes to write.

Throws:

`IOException` - if an I/O error occurs.

.

2.2 Anexa 1.

```
//Fişierul ClientFTP.java
import java.net.*;
import java.io.*;

//Clasa modeleaza un client FTP simplu.
public class ClientFTP
{
 //Metoda principala a clasei.
 public static void main( String args[] )
 {
    new ClientFTP();//aici doar apelez constructorul clasei
 }

 //Constructorul. Aici se petrec toate ...
 public ClientFTP()
 {
   String linie=null;//aceasta va fi o linie din fisierul text
deschis
   try{
       /*
```

```
          Folosnd un obiect URL, initializez o conexiune cu
serverul
          solicitand fisierul test.txt din radacina.
        */
                    URL            url            =            new
URL("ftp://admin:1234@80.96.123.147/test.txt");

        /*
          Creez un flux de date de citire:
          directionat de la server la client
        */
        InputStream is = url.openStream();

        /*
          Creez un obiect reader de tip BufferedReader,
          pe care il voi folosi
         pentru a citi text din fluxul de intrare,
          linie cu linie
        */
        BufferedReader br =
           new BufferedReader( new InputStreamReader( is ) );

        //incep citirea intr-o bucla
        while( true )
        {
          //incerc sa citesc o linie
          linie = br.readLine();

          //daca nu am reusit sa citesc din buffer
          if( linie == null )
            break;//gata

          //daca am reusit, afisez linia citita
          System.out.println( linie );
        }

    }catch( Exception e )
      {
        System.err.println( "Eroare: " + e );
      }
  }
}
```

LABORATOR 2

Programarea folosind BSD sockets

Referinte

Pentru a invata modul de utilizare a socket-urilor in C si Java, puteti sa consultati urmatoarele materiale:

- Curs
- UNIX Sockets Tutorial
- Java Networking Tutorial

Observatie: *Dupa cum reiese si din materialele referite mai sus, este important de remarcat ca primitivele numite "socket" in Java sunt destul de diferite de corespondentele lor din C. De fapt, Java abstractizeaza socket-urile reale, punand la dispozitia utilizatorului stream-uri prin care se poate lucra in modul obisnuit din Java.*

Tipuri de servere

Paradigma cea mai folosita in aplicatiile care folosesc sockets este cea introdusa odata cu arhitectura *client-server*. Serviciile sunt oferite de calculatoare pe care ruleaza componente de tip *server*, fiind folosite de catre unul sau mai multi *clienti*.
Din punctul de vedere al modului in care accepta conexiunile de la clienti, exista doua modalitati de a proiecta un server. Alegerea metodei se face tinand seama de cerintele sistemului software ce se doreste a fi implementat si de considerente de eficienta a acestuia.

1. **Server iterativ**
   Un server iterativ poate deservi un singur client la un moment dat. Daca in timp ce un client este deservit, apar alte cereri de conexiune, acestea vor fi puse intr-o coada de asteptare. Este posibil ca la un moment dat coada sa se umple, iar noile cereri sa fie respinse.

   Secventa urmatoare de (pseudo)cod arata modul de implementare in C a unui astfel de server:

```
int sockfd, newsockfd;
if ((sockfd=socket(...)) < 0)
  {
    printf ("error ..."); exit(1);
  }

if (bind(sockfd,...) < 0)
  {
    printf ("error ..."); exit(1);
  }

if (listen(sockfd,5) < 0)
  {
    printf ("error ..."); exit(1);
  }

for (;;)
```

```
{
 newsockfd = accept(sockfd, ...); /* blocare pentru
asteptare cereri*/
 if (newsockfd < 0)
   {
     printf ("error ..."); exit(1);
   }

  process(newsockfd); /*deservire cerere*/

  close (newsockfd);
}
```

Se observa ca o noua cerere nu este acceptata pana cand cererea in curs nu a fost deservita.

2. **Server concurent**

Un server concurent este scris in asa fel incat poate deservi mai multi clienti la un moment dat. Principiul este simplu, si se bazeaza pe crearea cate unui proces fiu pentru deservirea fiecarui client. Serverele concurente sunt cele mai raspandite modalitati de oferire a serviciilor in retea.

Programul de mai jos arata un exemplu de astfel de server:

```
int sockfd, newsockfd;

if ((sockfd=socket(...)) < 0)
  {
     printf ("error ..."); exit(1);
  }

if (bind(sockfd,...) < 0)
  {
     printf ("error ..."); exit(1);
  }

if (listen(sockfd,5) < 0)
  {
     printf ("error ..."); exit(1);}
  }

for (;;)
{
 newsockfd=accept(sockfd, ...); /* blocare pentru
asteptare cereri*/

 if (newsockfd < 0)
   {
     printf ("error ..."); exit(1);
   }

 if (fork()==0)  /* procesul fiu */
```

```
    {
      close(sockfd);
      process(newsockfd); /*procesare cerere*/
      exit(0);
    }

  close (newsockfd);  /* parinte */
}
```

Se observa ca atat parintele cat si fiul inchid socket-ul de care nu mai au nevoie: parintele inchide noul descriptor primit dupa accept() pentru ca el nu doreste sa trimita sau sa primeasca date prin intermediul sau, iar fiul inchide socket-ul initial, deoarece nu asteapta conexiuni noi din retea.
Daca este necesar, se pot introduce metode de limitare a numarului de clienti care se pot conecta, prin conditionarea prin program a efectuarii apelului accept() de numarul de clienti deja conectati.

LABORATOR 3

Programarea aplicaţiilor bazate pe protocoale

Consideraţii teoretice

1 Programarea cu socketuri. Concepte.

Există diverse tipuri de socketuri, diferenţiate după topologia comunicaţiei respectiv a tipurilor de adrese utilizate şi anume adrese Internet (în mod corespunzător socketurile se numesc socketuri Internet), căi către noduri locale pentru comunicarea între procese situate pe aceeaşi maşină (socketuri Unix) sau adrese tip X.25 (socketuri X.25, mai puţin utilizate).

Cateva **concepte specifice** implementării mecanismelor de comunicaţie de date utilizând socketurile vor fi detaliate în cele ce urmează.
Socket - reprezintă un capăt de comunicaţie, definind o formă de comunicare între procese (Inter Process Communication) situate pe aceeaşi maşină sau pe maşini diferite conectate printr-o reţea. O pereche de socketuri conectate realizează o interfaţă de comunicaţie între două procese, interfaţă similară celei de tip pipe din Unix.

Asocierea- defineşte în mod unic conexiunea stabilită între două capete de comunicaţie, ca fiind un set de parametri de foma *(protocol_local, adresa _locală, port_local, adresa_remote, port_ remote).* Noţiunile de *port_sursă*, respectiv *port_ destinaţie* permit definirea unică a comunicaţiei între perechi de procese şi realizează identificarea unică a unui SAP (Service access point) de nivel transport.

Descriptor de socket - reprezintă un descriptor de fişier, este argumentul utilizat în funcţiile de bibliotecă.

Binding – operaţia prin care se asociază o adresă de socket unui socket, pentru ca acesta să poată fi accesat.

Portul – reprezintă un identificator folosit pentru a diferenţia socketurile aflate la aceeaşi adresă. Pot fi folosite adrese cuprinse între 1024- 49151 (porturi înregistrate), respectiv adrese între 49152 – 65535,numite şi porturi efemere sau dinamice, adresele între 1 şi 1023 sunt rezervate de sistem(vezi fişierul etc/services din UNIX) .

Familie de adrese - reprezintă formatul de adrese folosit pentru a identifica adresele utilizate în socketuri(cele mai uzuale forme de adrese sunt adresele Internet şi Unix).

Adresa de socket - este funcţie de familia de adrese utilizate (pentru domeniul familiei de adrese Internet constă din adresa Internet şi adresa portului utilizat în host.

Adresa Internet – reprezintă o adresă pe 4 octeţi care identifică un nod (host sau ruter) într-o reţea.

2. Modelul client-server

Orice operaţie în reţea poate fi privită ca un proces client care comunică cu un proces server. Procesul server creează un socket, îi asociază o adresă şi lansează un mecanism de ascultare a cererilor de conectare din partea clienţilor, procesul client crează la rândul său un socket şi solicită conectarea la server. După acceptarea cererii de către server şi stabilirea conexiunii, se poate stabili comunicaţia între socketuri.

Acest model clasic poate varia în funcţie de natura aplicaţiei proiectate, astfel el poate fi simetric sau asimetric, bazat pe comunicaţie simplex sau duplex. Bazat pe acest model client-server au fost proiectate o serie de aplicaţii uzuale şi anume `telnet (pentru conectarea la un host` remote folosind portul 23) pentru care un program aflat în acel host, respectiv `telnetd`, va răspunde, `ftp/ftpd`, sau `bootp/bootpd`.

Lista funcţiilor uzuale din biblioteca de socketuri:

- `socket()` - creează un nou descriptor de socket;
- `bind()` – realizează legarea unei adrese şi a portului corespunzător la socket;
- `connect()` - permite stabilirea unei conexiuni active cu un server remote
- `listen()` – ascultă cererile de conectare, este utilizată într-un *socket pasiv;*
- `accept()` – permite crearea unui nou socket corespunzător unei noi cereri de conectare;
- `send(),recv(),sendto(),recvfrom()`– transmite- recepţionează *streamuri* sau *datagrame;*
- close(), shutdown()- închide o conexiune;
- `getpeername()` – determină numele hostului pereche din conexiune;
- `getsockname()` – determină numele socketului utilizat;
- `gethostbyname(), gethostbyaddr()`- deteremină numele sau adresa socketului din hostul specificat.

Comunicarea între procesele client, respectiv server se bazează pe apelul funcţiei `socket()`, apel care returnează un descriptor de socket. Acest descriptor este utilizat în apelurile diferitelor funcţii specializate pentru transmisia de date (cum ar fi de exemplu `send()` şi `recv()). Detalii de sintaxa pentru aceste funcţii se găsesc în anexa 1.`

3. Socketuri de tip stream vs. socketuri datagramă

Comunicaţia între programele client şi server se desfăşoară folosind protocoale de nivel *transport orientate conexiune (TCP), sau protocoale neorientate conexiune (UDP).*

Stream socketurile reprezintă fluxuri de comunicaţie sigure (fără erori), de tip full-duplex, bazat pe protocolul TCP ("Transmission Control Protocol") care asigură transmisia secvenţială şi fără erori a datelor.

Socketuri datagramă – sunt cunoscute şi sub numele de socketuri fără conexiune, deoarece ele folosesc adresele IP pentru rutare, dar protocolul de nivel transport utilizat este UDP("User Datagram Protocol"). Aceste socketuri nu menţin o conexiune deschisă pe durata comunicaţiei, ele realizând transferul pachet cu pachet (exemple de aplicaţii care utilizează socketurile datagramă sunt tftp (Trivial File Transfer Proticol), bootp ().

4 Tipuri de date utilizate de interfaţa de socketuri

Structura următoare memorează adresa de socket pentru diferite tipuri de socketuri şi este folosită în practica programării:

```
struct sockaddr
{
unsigned short sa_family; /* familia de adrese, AF_xxx */
char sa_data[14];         /* 14 bytes adresa de protocol */
};
```

Pentru a putea utiliza în programe structura `struct sockaddr`, a fost proiectată o structură paralelă: `struct sockaddr_in` definită în continuare astfel:

```
struct sockaddr_in
{
short int  sin_family; /*familia de adrese */
unsigned short int  sin_port;    /* număr port */
struct in_addr sin_addr;    /* adresă internet */
char sin_zero[8]; /* neutilizaţi */
};
```

iar

```
/* adresa Internet  */
  struct in_addr
  {
  unsigned long s_addr;
  };
```

Această structură permite o referire simplă a elementelor unei adrese de socket. Astfel `sin_zero` va fi setată la zero folosind funcţiile `bzero()` sau `memset()`. Pointerul la `struct sockaddr_in` mapează pointerul la structura `struct sockaddr`. Trebuie remarcat faptul că `sin_family` corespunde la `sa_family` în structura `struct sockaddr şi va fi setată la "AF_INET"` valoare utilizată pentru familia de protocoale TCP-IP.

5. Funcţii de conversie

Deoarece în transmisia de date, pentru a proiecta aplicaţii portabile este necesară realizarea conversiei datelor transmise între hosturi care utilizează în mod diferit fie reprezentarea little_endian,fie cea big_endian şi deoarece reprezentarea datelor la nivel de reţea este unică, (protocoalele din familia TCP-IP folosesc reprezentarea la nivel de reţea numită *Network Byte Order*) sunt necesare o serie de funcţii care să realizeze aceste conversii.

Câteva din cele mai utilizate funcţii de conversie în programarea cu socketuri sunt:

- `htons()`--"Host to Network Short", conversie host la reţea short;

- `htonl()`--"Host to Network Long",conversie host la reţea long;
- `ntohs()`--"Network to Host Short", conversie reţea la host short;
- `ntohl()`--"Network to Host Long", conversie reţea la host long.

`Cunoscut fiind faptul că sin_addr` şi `sin_port` fiind încapsulate în pachetul IP, respectiv UDP-TCP, este necesar să fie în ordinea Network_byte_order, iar câmpul `sin_family` utilizat fiind de către kernelul sistemului de operare (pentru a determina ce tip de adresă conţine structura), este în ordinea Host Byte Order, este necesară utilizarea funcţiilor de conversie.

Presupunem că analizăm structura `struct sockaddr_in ina`, şi o adresă IP "xxx.yyy.z.uu" pe care dorim să o memorăm în structură. Funcţia pe care o utilizăm este, `inet_addr()`, converteşte o adresă IP din forma cunoscută (notaţia numbers-and-dots) într-un întreg fără semn astfel:

```
ina.sin_addr.s_addr = inet_addr("xxx.yyy.z.uu");
```

`inet_addr()` returnează adresa în Network Byte Order.Pentru tipărirea unei adrese în notaţia cunoscută cu punct, este necesară utilizarea funcţiei `inet_ntoa()` ("ntoa" reprezintă "network to ascii") aşa cum se poate remarca în exemplul :

```
printf("%s",inet_ntoa(ina.sin_addr));
char *a1, *a2;
   .
   .
a1 = inet_ntoa(ina1.sin_addr);
a2 = inet_ntoa(ina2.sin_addr);
printf("address 1: %s\n",a1);
printf("address 2: %s\n",a2);
```

Mersul Lucrarii

Lucrarea îşi propune prezentarea noţiunilor care stau la baza mecanismelor transmisiei de date sub formă de mesaje între procese tip client - server bazată pe socketuri (Unix sau în Internet.) cât şi prezentarea diverselor modalităţi de comunicare client – server, asimilarea funcţiilor principale de bibliotecă (socketurile Unix, sau Internet) utilizate în proiectarea aplicaţiilor de tip client – server.

3.1 Rulaţi programul `listener` pe o maşină, apoi `talker` pe alta şi urmăriţi modul în care se desfăşoară comunicaţia. Programul `talker` apelează funcţia `connect()` şi specifică adresa `listener`-ului pentru a trimite şi recepţiona. Codul sursă pentru programele de test se află la sfârşitul lucrării. Modificati-l corespunzator astfel încat sa se poata desfăşura comunicaţia în situaţia în care ambele programe rulează pe aceeaşi maşina fizică repectiv pe masini diferite.
3.2. Proiectaţi trei programe de test *client1.c, client2.c şi server.c* conectate într-un pipeline logic, astfel: programul client1.c transferă un mesaj de tip caracter procesului server.c care

recepţionează caracterul, realizează o prelucrare asupra lui şi transmite procesului client2.c care tipăreşte rezultatul.
Modificaţi programele astfel încât valoarea transferată prin pipeline să fie consecutiv un întreg, respectiv o structură conţinând valori de tip caracter şi întreg.
3.3. Modificaţi programele proiectate anterior astfel încât să poată *funcţiona şi în domeniul Internet*.Dacă arhitecturile maşinilor din domeniu sunt diferite, deci structurile pentru mesaje sunt de dimensiuni diferite, proiectaţi un algoritm pentru a putea transmite structura.
3.4. Să se proiecteze un server simplu (de tip multiplexor), care să permită unui sistem de trei clienţi conectaţi într-o topologie de tip stea să comunice între ei. Programul server acceptă conexiuni din partea oricărui client şi va stabili conexiunile de tip socket cu fiecare dintre clienţi. Serverul va tipări mesajul recepţionat şi apoi îl va transmite clientului destinaţie. Transmiterea mesajului (cu lungimea de 40 caractere) se va realiza la introducerea caracterului CR. Mesajul va conţine identificatorii clienţilor sursă respectiv destinaţie (ID), iar clientul care recepţionează mesajul îl va scrie în fişierul propriu de mesaje. Pentru fiecare client se va proiecta un meniu care să permită selecţia uneia dintre opţiunile: alegere client cu care comunică, tipărire-ştergere mesaje curente aflate în fişierul de mesaje al clientului, ieşire cu ştergerea nodului-client din reţea.
3.5. Să se proiecteze un server de timp. Serverul trebuie să poată răspunde solicitărilor simultane a mai multor clienţi, consultând o bază de date care conţine data şi ora exactă pentru zona solicitată.
3.6. Proiectaţi un sistem de tip client server care să permită transferul unui fişier de pe maşina server pe cea client.

RPC

Introducere:

RPC (Remote Procedure Call) - extinde modelul obisnuit al apelurilor de proceduri valabil pe un calculator local permitind ca diversele componente ale unei aplicatii sa ruleze pe calculatoare distincte. Acest model permite o tratare la nivel inalt a sistemelor client-server deoarece diferitele componente ale aplicatiei (functii) se vor executa pe alte calculatoare, *acest nivel de abordare fiind superior utilizarii socket-urilor*.

Suport teoretic:

In codul *clientului* apelurile de functii pentru server vor trece prin asa numitele *stubs* (schelete) de functii client, pentru programul principal aceste schelete fiind adevaratele functii. In codul *serverului* exista functii analoage scheletelor din client numite *wrappers* (invelitori). Nivelul de programare retea este ascuns in spatele acestor functii degravand programatorul de utilizarea socketurilor sau implementarea unor protocoale de transmitere al parametrilor si de returnare al rezultatului.

Majoritatea implementarilor RPC au o serie de *restrictii*:

- numar parametrii
- modul de returnare al rezultatului

Fiecare implementare RPC include urmatoarele *componente*:

- un *"limbaj"* pentru specificarea identitatii procedurilor apelabile si formatul argumentelor si al valorii returnate
- un *instrument* care sa genereze automat stuburile si wrapperele
- *set de reguli* care sa defineasca formatul datelor. Exemplu: *XDR* (External Data Representation)
- un *serviciu de inregistrare* care sa permita sa foloseasca serverul dorit (exemplu: portmapper)
- o *biblioteca de functii* care sa trateze detaliile de comunicare, serializare/deserializare (existenta pe SunOS sau Solaris si la nivelul nucleului pe Linux)

NOTA: In cele ce urmeaza toate exemplele vor trata un exemplu concret de program ce furnizeaza prin mecanismul RPC ora curenta.

Identificarea unui serviciu RPC se face pe baza unei adrese formate din: ***{numar program, numar versiune, numar procedura}***.

date.x - exemplu de fisier de specificare al protocolului:

```
program DATE_PROG {
   version DATE_VERS {
      long   BIN_DATE(void) = 1;
      string STR_DATE(long) = 2;
   } = 1;
} = 0x31234567;
```

Utilitarul (compilatorul) rpcgen se foloseste in urmatorul mod: **rpcgen date.x**. Vor rezulta urmatoarele fisiere (sub Linux):

- date_svc.c - cuprinde wrapperele pentru fiecare functie si o functie main.
- date_clnt.c - cuprinde stuburile pentru fiecare functie.
- date.h - fisier antet ce contine declaratiile stuburilor si ale wrapperelor (diferite sub Linux).

Atentie: sub SunOS ar rezulta patru fisiere distincte fisierul in plus continand filtrele XDR.
Programatorul mai trebuie sa furnizeze doua fisiere:

- *rdate.c* (remote date) care implementeaza clientul si apeleaza procedurile la distanta ca:

```
if ( (lresult = bin_date_1 (NULL, cl)) == NULL) {
    clnt_perror(cl, server);
    exit(3);
}
```

creerea handlerului de client *CLIENT *cl* facandu-se in prealabil cu:

```
/* create the client handle */
if ( (cl = clnt_create (server, DATE_PROG, DATE_VERS, "udp")) == NULL) {
   clnt_pcreateerror(server);
   exit(2);
}
```

acest handler va fii eliminat ulterior prin:

```
clnt_destroy(cl);
```

- *dateproc.c* cuprinde implementarile functiilor apelate la distanta, exemplu:

```
long* bin_date_1_svc(void* v, struct svc_req* req)
{
   static long timeval;
   long time();
   timeval = time((long *)0);
   return(&timeval);
}
```

Atentie: in ambele fisiere trebuie incluse *<rpc/rpc.h>* si *"date.h"*.

Mersul lucrarii:

Se va implementa folosind exemplele date un program client -server care sa afiseze data curenta existenta pe masina pe care ruleaza serverul pe masina pe care ruleaza clientul. Vor fi apelate la distanta doua functii *bin_date_1* si *str_date_1* si care vor fi implementate in server ca bin_date_1_svc si str_date_1_svc. Prima din functii furnizeaza timpul in format *long* (apel *time()*) iar cea de a doua transforma acest *long* in *char** (apel functiei *ctime()*).

LABORATOR 7

Utilizarea socket-urilor

Introducere:

In lucrarea de fata se va studia lucrul cu *socket*-uri in Java si din acest motiv se vor studia in plus si alte clase de intrare iesire ce usureaza mult lucrul cu *stream*-urile.

Suport teoretic:

Lucrul cu socket-uri sub Java seamana extrem de mult cu utilizarea acestora sub Unix. In cadrul lucrarii vor fi folosite urmatoarele clase:

- din pachetul *java.net*:
  - *InetAddress* - se pot crea adrese Internet
  - *ServerSocket* - socket pe partea de server
  - *Socket* - socket pe partea de client
- iar din pachetul *java.io*:
  - *BufferedInputStream*
  - *BufferedOutputStream*
  - *BufferedReader*
  - *BufferedWriter*
  - *DataInputStream*
  - *DataOutputStream*
  - *InputStream*
  - *InputStreamReader*
  - *PrintStream*

Exemplu simplu de implementare al unui server:

```
import java.io.*;
import java.net.*;
class EchoServer {
   public static void main(String args[]) {
      ServerSocket server = null;
      Socket sock = null;
      DataOutputStream os = null;
      BufferedReader is = null;
      String line;
      boolean up = true;
      boolean more;

      try {
         server = new ServerSocket(4444);
      } catch(IOException e) {
         System.out.println(e);
      }

      try {
         while(up) {
            //accept
            sock = server.accept();
            is = new BufferedReader(
```

```
                        new
InputStreamReader(sock.getInputStream()));
            os = new
DataOutputStream(sock.getOutputStream());

            System.out.println("New client accepted");

            more = true;
            while (more && up) {
               line = is.readLine();
               System.out.println("Receive: " + line+
"\n");
               more = (line.compareTo("bye") != 0);
               up = (line.compareTo("down") != 0);
               os.writeChars("EchoServer responde: " +
line + "\n");
            }

            os.close();
            is.close();
            sock.close();
         }//up
         server.close();

      } catch(IOException e) {
         System.out.println(e);
      }
   }
}
```

A se observa modul cum se creaza mai intai o clasa ServerSocket si in urma acceptarii unei cereri se creaza un Socket de comunicatii cu clientul. Ca diferenta fata de modul de abordare din Unix se observa deschiderea unor stream-uri io cu care se lucreaza de fapt in cadrul aplicatiei. Acest exemplu poate fi complilat si testat cu telnet-ul.

Pe partea de client ca exemplu putem avea:

```
import java.io.*;
import java.net.*;
class EchoClient {
   public static void main(String args[]) {
      Socket sock = null;
      DataOutputStream os = null;
      BufferedReader is = null;
      BufferedReader cin = new BufferedReader(
                                 new
InputStreamReader(System.in));
      String line = null;

      try {
         System.out.println("Echo Client started");
         sock = new Socket("localhost", 4444);
```

```
            is = new BufferedReader(
                        new
InputStreamReader(sock.getInputStream()));
            os = new
DataOutputStream(sock.getOutputStream());

            while(true) {
               line = cin.readLine();
               os.writeChars(line + "\n");
               os.flush();
               System.out.println("Send to server: " + line
+ "\n");
               line= is.readLine();
               System.out.println("Receive from server: " +
line);
            }
         } catch (IOException e) {
         }
      }
}
```

Mersul lucrarii:

1. Se vor studia in mod obligatoriu (intr-o prima faza exhaustiv) intraga lista de clase prezentata in prima parte.
2. Se va compila si testa serverul cu ajutorul telnet-ului (a nu se omite modificarea portului de lucru al serverului).
3. Se va testa si clientul si se vor face modficarile necesare pentru compatibilitatea cu serverul (in acest scop se vor studia amanuntit clasele io prezentate).
4. La alegere se vor realiza urmatoarea lucrare:
   - server si client FTP
   - server si clienti de stiri

   respectand cerintele de la lucrarile de Unix.